

GOMS, GOMS+, and PDL

Michael Freed and Roger Remington
NASA Ames Research Center

Abstract: The expressiveness of the GOMS language for describing “how-to” knowledge determines what kinds of human activities can be captured by a GOMS task analysis. This paper addresses the adequacy of this framework for representing human behavior in realistically time-pressured, uncertain, and otherwise demanding task environments. Several improvements have been incorporated into a GOMS-like formalism called Procedure Description Language (PDL) and into a simple GOMS extension called GOMS+.

1 GOMS

Computer simulation has become an indispensable design aid for devices ranging from electronic circuits to automobile engines. Applying this technique to the design of human-machine systems would be desirable, but modeling the human components of these systems poses significant challenges. Researchers have addressed these by developing a variety of frameworks for human performance modeling. Among these, GOMS (Card, Moran, & Newell, 1984) has become the most well-known and widely used.

GOMS is a formal language for representing how human operators carry out specified routine tasks. It consists of four constructs: goals, operators, methods, and selection-rules (hence the GOMS acronym).

Goals represent desired actions or world states – e.g. *GOAL:(delete-file file-32)* expresses a desire to remove (access to) a computer file called file-32. GOMS interpreter¹ mechanisms determine behavior by mapping goal structures to sequences of **operators**, each representing a basic physical or cognitive skill such as shifting gaze to a new location, or grasping an object.

The GOMS approach assumes that the modeled agent has already learned one or more **methods** for accomplishing any goal. A method, available in a modeled agent’s “method library,” is used to decompose a goal into a sequence

¹ Interpretation rules that define GOMS semantics can be incorporated into a computer program, allowing simulation of a GOMS agent. Though the interpretation function is often carried out “by hand” (John and Kieras, 1994), longer, more complicated tasks require automation.

of subgoals and operators. For instance, the method below (specified informally) might be used to achieve the file deletion goal. Some steps of the method (e.g. steps 3 and 6) correspond to operators and can thus be executed directly. The others define subgoals that must be recursively decomposed using methods from the method library until an operator sequence is fully specified.²

Method-126 for goals of form (delete-file ?file)

1. determine ?location of ?icon for ?file
2. move-mouse-pointer to ?location
3. press-with-hand mouse-button
4. determine ?location2 of trash-icon
5. move-mouse-pointer to ?location2
6. release-hand-pressure mouse-button

When there is more than one way to achieve a goal – i.e. alternative methods are available – a **selection rule** must be used to decide between them. For instance, one alternative way to delete a file (call it method-823) may be to use a search utility designed to locate named files in a large file directory structure, and then use the utility's

delete option once the target file has been found. A selection rule for deciding between these methods might incorporate knowledge that method-126 is more convenient and therefore preferable as long as the location of the file is known. Thus:

Selection-rule for goal:(delete-file ?file)

IF known (location ?file)
THEN do method-126
ELSE do method-823

Nothing in the GOMS architecture specifies human behavioral characteristics that would naturally produce human-like behavior. GOMS could just as easily be used to describe the action-selection behavior of an insect, superhero or robot. For example, in GOMS, one could easily define a method for parking a car that involved picking the vehicle up and placing it in the desired location; nothing about GOMS makes this less desirable than a method more in line with typical behavior. Since many of the performance predictions one would wish to make with GOMS require a model of human attributes (e.g. strength limitations), GOMS is often coupled with a human attribute model such as the Model Human Processor (Card, Moran, & Newell, 1984). Efforts to make GOMS a more powerful tool have focused almost exclusively on

² Throughout this paper, a leading question mark will be used to indicate a variable.

refining the model of human attributes, particularly temporal attributes of behavior such as the time needed to press a button or shift gaze (Olson and Olson, 1989; John and Kieras, 1994). The GOMS language itself has remained unchanged.

The expressiveness of the GOMS language for describing “how-to” knowledge determines what kinds of human activities can be captured by a GOMS task analysis. In some ways, this language is clearly inadequate for describing even mundane behaviors. For example, people generally know to delay deciding which elevator door to move towards until there is some indication of where the next elevator will arrive. People’s ability to delay in this case illustrates a general and crucial human capability: to wait until information becomes available (uncertainties are resolved) before committing to a course of action. GOMS does not support this capability.

In this paper, we turn our attention to the adequacy of the GOMS notation for representing human behavior in task environments that are realistically time-pressured, uncertain, and otherwise demanding. The problem of characterizing agent capabilities needed for a given task environment has long been a focus

of research within subfields of artificial intelligence concerned with planning and plan execution. The resulting understanding can be roughly broken down into two main areas: (1) capabilities needed to cope with uncertainty; and (2) capabilities needed to manage multiple, interacting tasks. The following sections discuss how capabilities in these areas have been incorporated into a GOMS-like formalism called Procedure Description Language (PDL). PDL was developed as part of the APEX human modeling framework, a tool for predicting usability problems in complex, dynamic task domains such as air traffic control. A subset of these improvements has been incorporated into GOMS+, a simple extension to GOMS that may appeal to APEX users for its simplicity and familiarity.

2 PDL

The central construct of the APEX Procedure Definition Language (PDL) is the **procedure**, closely analogous to the GOMS method³. The procedure consists of an **index clause** followed by one or more **step clauses**. The example procedure below represents how-to

³ A procedure generalizes the idea of method to include auxiliary activities such as post-completion behaviors (e.g. putting away the tools after a repair task), failure-handling actions (if turning the ignition key doesn’t work, try again) and interruption-handling actions (apologize when interrupting a phone conversation). The term **task** generalizes on goal in an analogous way.

knowledge for the routine behavior of turning on an automobile's headlights.

```
(procedure
  (index (turn-on-headlights)
    (step s1 (clear-hand left-hand))
    (step s2 (determine-loc
headlight-ctl => ?loc)
    (step s3 (grasp knob left-hand
?location)
      (waitfor ?s1 ?s2))
    (step s4 (pull knob left-hand)
(waitfor ?s3))
    (step s5 (ungrasp left-hand)
(waitfor ?s4))
    (step s6 (terminate) (waitfor
?s5))))
```

The index clause indicates that the procedure should be retrieved whenever a goal of the form (*turn-on-headlights*) becomes active. Step clauses primarily describe activities needed to accomplish the goal. Steps are assumed to be concurrently executable. For instance, step *s1* above for clearing the left hand (letting go of any held object) and step *s2* for locating the headlight controls do not need to be carried out in any particular order and do not interfere with one another; they may proceed in parallel. When order is required, usually because finishing one step is a precondition for starting another, this can be specified using a **waitfor** clause. For instance, one should not begin grasping the

headlight control until its location is known and the hand that will be used to grasp it is free. These preconditions are indicated by the waitfor clause embedded in step *s3*. The following two sections describe how the issues of uncertainty-handling and multitask management are addressed in PDL. Other aspects of PDL, including mechanisms for selecting between alternative procedures, managing periodic behavior, and adapting to varying degrees of time-pressure, are discussed in Freed (1998).

2.1 Coping with uncertainty

Some of the most important recent advances in AI concern how agents can act effectively in uncertain task environments. Early agent architectures were designed to operate in very simple environments in which all relevant aspects of the current situation (world state) are known, no change occurs except by the agent's action, and all of the agent's actions succeed all of the time. Most real-world domains are not so benign.

Uncertainty arises from a variety of factors. First, many real task environments are far too *complex* to observe all the important events and understand all the important processes. Decisions must therefore sometimes be made on the basis of guesswork about what is currently true and about

what will become true. Similarly, real task environments are often *dynamic*. Forces not under the agent's control change the world in ways and at times that cannot be reliably predicted. Previously accurate knowledge of the current situation may become obsolete when changes occur without being observed.

Additional sources of uncertainty arise from the nature of the agent itself. Motor systems may be clumsy and imperfectly reliable at executing desired actions. Perceptual systems may intermittently distort their inputs and thus provide incorrect characterizations of observed events. Cognitive elements may lose or distort memories, fail to make needed inferences, and so on.

Together, these various sources of uncertainty have a profound effect in determining what kinds of capabilities an agent requires to perform effectively. For example, the possibility that some action will fail to achieve its desired effect means that an agent needs some way to cope with possible failure. Thus, it may require specialized mechanisms that formulate explicit expectations about what observable effect its action should achieve, check those expectations against observed events, then, if expectations fail, generate new goals to recover, learn and try again.

PDL provides means for coping with several forms of uncertainty. The first, uncertainty about a future decision-relevant world state, is illustrated by the problem of deciding which elevator door to approach. A knowledgeable elevator user will usually delay committing to one door or the other until there is information about which will arrive next. This exemplifies a general strategy: delay decisions until relevant information becomes available. As seen in the procedure below, such strategies are represented using the *waitfor* clause.

```
(procedure
  (index (enter-first-available-
    elevator))
    (step s1 (summon-elevator))
    (step s2 (approach-elevator-door
      ?door)
      (waitfor ?s1 (open-door
        ?door)))
    (step s3 (enter-door ?door)
      (waitfor ?s2))
    (step s4 (reset ?self)
      (waitfor ?s2 (closed-door
        ?door)))
    (step s5 (terminate) (waitfor
      ?s3)))
```

In this procedure, the task of approaching an elevator door (step *s2*) does not begin until after an elevator has been summoned and its door has opened. When the latter event occurs, perceptual mechanisms detect an event of the

form (*open-door ?door*); this causes the variable *?door* to become bound to a mental representation of the newly opened door, thus resolving uncertainty about which elevator should be selected.

Waiting passively for new information to resolve uncertainty is one of three typical strategies. The others are actively seeking information and gambling (making a best guess). Active seeking is accomplished without any special PDL constructs. Procedures must simply initiate actions such as shifting gaze or physically removing a visual obstruction that result in new information becoming available. Gambling strategies can be encoded in several ways. The simplest is to incorporate a best guess directly into a procedure. For example, if the left elevator door is almost always the first to open, step *s2* in the procedure above might be replaced by:

*(step s2 (approach-elevator-door
left-door)
(waitfor ?s1))*

PDL and its interpreter support a variety of gambling strategies, each of which takes advantage of some kind of heuristic decision-making bias. For instance, assuming that the most frequent condition will hold in the current case (e.g. the left elevator appears

first usually, so it will this time as well) constitutes reliance on **frequency bias**. PDL supports this and other forms of bias⁴ with mechanisms that dynamically adjust how much influence bias has on a decision. Bias that is normally be weak and not relied upon may be strengthened in time-pressured situations or high-workload situations. Conversely, strong bias may be suppressed in some situations, especially in response to recently observed counterevidence. For instance, strong frequency bias may lead to a habit of approaching the left elevator in anticipation of its arriving first. But upon seeing a sign claiming that the left elevator is under repair, the agent may revert to a watch and see strategy. Heuristic biases are a pervasive and quite useful element of normal decision-making, but they can also lead to error (Reason, 1990). PDL support for representing biases makes it possible to represent cognitive mechanisms underlying error and thus facilitates error prediction (Freed and Remington, 1998).

PDL can also be used to represent knowledge about how to handle a second form of uncertainty that arises when actions

⁴ Frequency bias is a tendency to do or believe what is usually the case. **Recency bias** is a tendency to do or believe what was true last time. **Confirmation bias** is a tendency to do or believe what accords with one's expectations.

have more than one possible outcome. Multiple outcomes, particularly those constituting failure, require mechanisms for classifying the outcome and selecting an appropriate response. The procedure above provides a very simple example. Consider the case where, while trying to enter an open elevator door (step *s3*), the agent sees the door close. This should be construed as goal failure, causing the agent to try again. PDL provides an operator (low-level behavior) called **reset** for restarting (retrying) a task. Reset appears in step *s4*, and is invoked conditionally in response to seeing the elevator door close.

2.2 Managing multiple tasks

PDL and its interpreter were originally developed to represent the behavior of human air traffic controllers. As with many of the domains in which human simulation could prove most valuable, air traffic control consists mostly of routine activity; complexity arises primarily from the need to manage multiple tasks. For example, the task of guiding a plane to a destination airport typically involves issuing a series of standard turn and descent authorizations to each plane. Since such routines must be carried out over minutes or tens of minutes, the task of handling any individual

plane must be periodically interrupted to handle new arrivals or resume a previously interrupted plane-handling task.

The problem of coordinating the execution of multiple tasks differs from that of executing a single task because tasks can interact, most often by competing for resources. In particular, each of an agent's perceptual, motor and cognitive resources are typically limited in the sense that they can normally be used for only one task at a time. For example, a task that requires the *gaze* resource to examine a visual location cannot be carried out at the same time as a task that requires gaze to examine a different location. When separate tasks make incompatible demands for a resource, a *resource conflict* between them exists. To manage multiple tasks effectively, an agent must be able to detect and resolve such conflicts.

The PDL interpreter determines whether two tasks conflict by checking whether they both require control of a resource. Resource requirements for a task are undetermined until a procedure is selected to carry it out. For instance, the task of searching for a fallen object will require gaze if performed visually, or a hand resource if carried out by grope-and-feel. PDL denotes a procedure's resource requirements using the **profile** clause. For

instance, adding the clause (*profile* (*left-hand* 8 10)) to the turn-on-headlights procedure declares that turning on headlights conflicts with any other task that requires the left-hand.⁵

To resolve a detected resource conflict, decision-mechanisms must determine the relative priority of competing tasks, assign control of the resource to the winner, and either shed, defer, or interrupt the loser. To compute relative priority, the interpreter uses information provided in **priority** clauses. The simple form of a priority declaration specifies a numeric priority value ranging from 1 to 10. Alternately, a priority clause can specify **urgency** and **importance** values related to a specified source of priority.

```
(step s3 (enter-door ?door)
(waitfor ?s2)
(priority miss-elevator :urg 6
:imp 2))
```

For example, the step above for entering an elevator derives priority from the possibility that the opportunity to board will be missed if the task is delayed too long. Urgency is fairly high to denote a limited window of opportunity. Since missing the elevator is

usually far from catastrophic, importance is low.

In realistically demanding environments, several additional factors need to be considered in determining priority. For instance, a task's urgency and importance may be context-dependent (not constant-valued), and may in fact vary dynamically over the lifetime of a task. PDL allows users to specify how these values should be computed and under what circumstances they should be recomputed. A task may have several associated priority clauses, reflecting separate reasons to do the task sooner rather than later. For instance, the need to enter an elevator expeditiously could stem from a desire to get in before it closes and also a desire to get in ahead of others.

The possibility that a task might be interrupted presents additional issues. First, handling an interruption often entails carrying out transitional behaviors. For instance, interrupting a driving task typically involves doing something to keep from crashing such as pulling over to the side of the road. To facilitate such transitions, the PDL interpreter generates an event of the form (*suspended* <task>) whenever <task> is interrupted. The step below within the body of the driving procedure would produce this behavior:

⁵ The profile clause and all other PDL constructs related to multitask management are discussed in detail in (Freed, 1998b).

(step s15 (pull-over)
(waitfor (suspended ?self)))

Second, continuity bias, the tendency to continue executing an ongoing task rather than switch to an interrupting task, is represented with an **interrupt-cost** clause. We currently assume that the degree of continuity bias depends on objective factors that make an interruption costly – e.g. having to engage in otherwise unnecessary transition behaviors, having to make up lost progress on the main task, and so on. Interrupt cost raises a task's importance, and thus it's priority. Unlike a priority clause, which applies whenever a task is eligible to be executed, interrupt-cost only applies to ongoing tasks.

Third, it should be possible to take advantage of slack-time in a task's need for given resources. For example, when stopped behind a red light, a driver's need for hands and gaze is temporarily reduced, making it possible to use those resources for other tasks. Such within-procedure resource control strategies are specified using the **suspend** and **reprioritize** operators. Suspend allows a procedure to interrupt itself, relinquishing control over resources and thus making them temporarily available to lower priority tasks. Reprioritize causes the primary task to resume

competing for resources, normally resulting in its acquiring them from tasks active during the slack interval. E.g.:

(step s18 (reprioritize)
(waitfor (color ?traffic-light
green))))

would allow a driving task that self-suspended in response to a red traffic light to resume.

3 GOMS+

For APEX users more comfortable using GOMS than PDL, we have developed a GOMS implementation called GOMS+ that incorporates several of the capability extensions discussed in this paper. As with GOMS, methods in GOMS+ are action sequences. Behaviors that are contingent or off critical-path (such as those needed to handle failure) cannot be represented.

(method 1 for (turn-on-
headlights))
(requires left-hand)
(do-in-parallel
(clear-hand left-hand)
(determine-loc headlight-
control => ?loc))
(grasp knob left-hand ?loc)
(pull knob left-hand)
(ungrasp left-hand))

```
(method 1 for (enter-first-
available-elevator)
  (summon-elevator)
  ((approach-door) (waitfor
(open-door ?door))
  (enter-door ?door))
```

The GOMS+ methods above illustrate several features of the language. The construct **do-in-parallel** enables some concurrent behavior. For example, the clear-hand and determine-location steps of the turn-on-headlight method are concurrently executable. Some versions of GOMS already allow concurrent action, but only between operator-level behaviors; steps declared within the scope of a do-in-parallel clause can represent either operators or non-operators that will have to be decomposed into subgoals. GOMS+ includes a second construct for concurrency control called **race**. Steps declared within a race clause are carried out in parallel until any one step completes; at that point, the rest are forced to completion (aborted).

GOMS+ also adopts the **waitfor** clause. Since steps of method are sequentially ordered by default, there is no need to use waitfors to declare step order; in all other ways, the clause is used as a generic precondition declaration, just as in PDL. As previously noted, this is particularly useful for delaying actions until information

becomes available to resolve decision-relevant uncertainty.

The decision not to allow representation of contingent behaviors in GOMS+ means that sophisticated multitask management capabilities (which rely heavily on such behaviors) cannot be implemented. However, two multitasking constructs have been included. The **priority** clause assigns a fixed numeric priority to a method step and is used to choose between two simultaneously active goals (cf. John, Vera, and Newell, 1990). Note that this condition only occurs within the dynamic scope of a do-in-parallel/race clause or when multiple initial goals have been asserted. The **requires** clause declares that a method requires some resource such as the left hand (see turn-on-headlights procedure above). Actions that would otherwise be executable in parallel but require the same resource must be carried out sequentially. If the conflicting tasks have priority values, the higher valued task is done first; otherwise order is determined randomly.

Finally, GOMS+ includes the **repeat-until** clause for representing certain repetitive behaviors. This is mainly useful where goals are achieved by the cumulative effect of repetitive action. For example, one might repeat a dig action until a hole of

specified depth has been created or repeat a stirring action until food obtains a desired consistency. Certain forms of repetitive behavior require careful multitask management and therefore cannot be represented in GOMS+; these include especially maintenance behavior such as periodically scanning instruments on a flight deck or dashboard to maintain situation awareness. Other forms are being considered for inclusion in the language including **repeat-times** to repeat an action a specified number of times, and **repeat-at-interval** to cause an action to repeat after a specified amount of time has passed.

4 Conclusion

GOMS is essentially a special-purpose programming language for specifying agent behavior. Evaluating its effectiveness in this regard means asking a set of questions well-known to programming language users and designers. For example, does it have a clear and unambiguous semantics? Is it elegant? Intuitive? But most important is: can desired behaviors be expressed effectively and conveniently? A well-designed language anticipates the uses to which it will be put by providing terminology and structure for those uses. For GOMS, that means providing constructs to represent

common elements of intelligent behavior.

In its original form, GOMS incorporates three such elements: the use of predefined action sequences to achieve common goals, the ability to select between alternative action sequences, and the ability to define goals at varying levels of abstraction, entailing recursive decomposition into subgoals. If one could choose to incorporate only 3 facets of intelligent behavior in a model, perhaps these would be the best choices.

In our view, there is no reason to employ so parsimonious a language. Extreme simplicity makes it easy to interpret GOMS notations; but this virtue recedes in importance once the interpretation process has been incorporated into a computer program. Ease of learning trades off against inexpressiveness if users have to spend time struggling to circumvent the language's limitations. Human behavior is rich and varied, especially in the realistically demanding task environments where human performance modeling could be most valuable. With PDL and GOMS+, we hope to make it possible and practical to construct such models.

5 References

Card, S.K., Moran, T.P., & Newell, A. (1983). *The psychology of human-computer interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Freed, M. (1998a) Simulating human performance in complex, dynamic environments. Ph.D. Dissertation, Department of Computer Science, Northwestern University.

Freed, M. (1998b) Managing multiple tasks in complex, dynamic environments. In Proceedings of the 1998 National Conference on Artificial Intelligence. Madison, Wisconsin.

Freed, M. and Remington, R. (1998) A conceptual framework for predicting error in complex human-machine environments. Proceedings of the 20th Conference of the Cognitive Science Society. Madison, Wisconsin.

Gray, W. D., John, B. E., Atwood, M.E. (1993). Project Ernestine: Validating a GOMS analysis for predicting and explaining real-world task performance. *Human Computer Interaction*, **8**, 237-309.

John, B.E. and Kieras, D.E. (1994). *The GOMS Family of Analysis Techniques: Tools for Design and Evaluation*. Carnegie

Mellon University, TR CMU-CS-94-181.

John, B.E. and Vera, A. (1992) A GOMS analysis of a graphic, machine-paced, highly interactive task. Proceedings CHI'92, ACM, 251-258.

Olson, J.R. and Olson G.M. (1989) The growth of cognitive modeling in human-computer interaction since GOMS. *Human Computer Interaction*.

Reason, J.T. (1990) *Human Error*. Cambridge University Press, New York, N.Y.